



O modalitate de învățare a metodei backtracking

prof. Doru Popescu Anastasiu, Slatina

“În memoria lui Tudor Sorin.”

Rezumat

În urmă cu aproximativ 16 de ani Tudor Sorin propunea lumii informatice din România un șir de materiale prin care standardiza metoda backtracking. Acest lucru a dus la crearea unui șablon prin care o metodă putea fi folosită deopotrivă de către elevi, studenți, profesori. În continuare voi prezenta această metodă de programare ca o continuare firească a capitolului “Recursivitate” din clasa a X-a, profil matematică-informatică, intensiv informatică.

Introducere

Mulți profesori încheie capitolul recursivitate cu probleme de tipul:

Se dă un număr natural $n < 20$ și un alfabet cu două litere ($A = \{a, b\}$). Determinați toate cuvintele de n litere folosind alfabetul dat.

Exemplu

cuvinte.in	cuvinte.out
2	aa ab ba bb

Rezolvarea problemei fiind următoarea:

Un cuvânt poate fi memorat într-un vector x cu n componente de tip `char` (fiecare componentă putând să aibă una din valorile ‘a’, ‘b’). Recursiv, cuvintele se pot determina folosind programul următor:

```

var x:array[0..21] of char;
    n:integer;
    f:text;

procedure afisare;
var i:integer;
begin
  for i:=1 to n do
    write(f,x[i]);
  writeln(f);
end;

#include <fstream.h>
char x[21];
int n;
ofstream fout("cuvinte.in");

void afisare(){
  int i;
  for(i=1;i<=n;i++)
    fout<<x[i];
  fout<<'\\n';
}
```

```

procedure generare(k:integer);
begin
if k=n+1 then
  afisare
else
  begin
    x[k]:='a';
    generare(k+1);
    x[k]:='b';
    generare(k+1);
  end
end;

begin
  assign(f, 'cuvinte.in');
  reset(f);
  read(f,n);
  close(f);
  assign(f, 'cuvinte.out');
  rewrite(f);
  generare(1);
  close(f)
end.

```

```

void generare(int k){
  if(k==n+1)
    afisare();
  else
  {
    x[k]='a';
    generare(k+1);
    x[k]='b';
    generare(k+1);
  }
}

int main(){
  ifstream fin("cuvinte.in");
  fin>>n;
  fin.close();
  generare(1);
  fout.close();
  return 0;
}

```

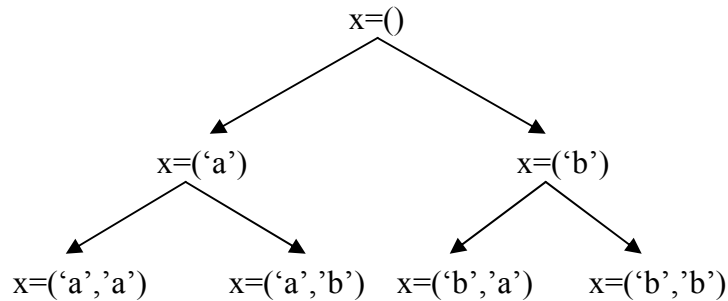
Pentru exemplul dat în enunț șirul de apeluri este următorul:

```

generare(1) → generare(2) → generare(3)
x[1]='a'      x[2]='a'
                                     se stopează generarea prin condiția k=n+1 și
                                     se afișează aa
                                     ←
                                     x[2]='b'
                                     → generare(3)
                                     se stopează generarea prin condiția k=n+1 și
                                     se afișează ab
x[1]='b' ←
          → generare(2) → generare(3)
          x[2]='a'      se stopează generarea prin condiția k=n+1 și
                               se afișează ba
                               ←
                               x[2]='b'
                               → generare(3)
                               se stopează generarea prin condiția k=n+1 și
                               se afișează bb
stop ←

```

Se obține astfel următorul arbore, cu evoluția vectorului x:



Generalizări

Ne punem în continuare problema, cum putem determina cuvintele care folosesc un alfabet mai mare (de exemplu cu primele m litere din alfabetul englez). Mai mult, cum determinăm cuvintele care îndeplinesc o anumită proprietate (de exemplu: să nu aibă două litere la fel pe poziții consecutive) ș.a.m.d.

Dacă ne concentrăm pe problema determinării cuvintelor, folosind primele m litere din alfabetul englez cu restricția: *să nu existe litere egale pe poziții consecutive*, o variantă de subprogram ce generează vectorul x ar putea fi:

<pre>procedure generare(k:integer); var i:integer; begin if k=n+1 then afisare else for i:=1 to m do begin x[k]:=chr(ord('a')+i-1); generare(k+1) end end end; end;</pre>	<pre>void generare(int k){ int i; if(k==n+1) afisare(); else for(i=1;i<=m;i++) { x[k]='a'+i-1; generare(k+1); } }</pre>
---	--

Menționăm faptul că, în funcția de afișare se va verifica condiția ca să nu existe două litere egale pe poziții consecutive, afișarea făcându-se numai în acest caz.

În aceste condiții pentru $n=2$ și $m=2$ se generează același arbore ca mai sus. Unele dintre ramuri fiind generate degeaba, pentru că odată construite două componente cu indici consecutivi și aceeași literă, sigur nu se va ajunge la un cuvânt ca să fie afișat. Astfel pentru a eficientiza algoritmul de generare se impune adăugarea unei condiții la fiecare pas (legată de alegerea valorii pentru componenta $x[k]$, adică $x[k] \neq x[k-1]$).

Obținem astfel rafinarea:

```

procedure generare(k:integer);
var i:integer;
begin
if k=n+1 then
  afisare
else
  for i:=1 to m do
  begin
    x[k]:=chr(ord('a')+i-1);
    if conditie(k) then
      generare(k+1)
    end
  end;
end;

```

```

void generare(int k) {
int i;
if (k==n+1)
  afisare();
else
  for (i=1; i<=m; i++)
  {
    x[k]='a'+i-1;
    if (conditie(k))
      generare(k+1);
  }
}

```

Funcția condiție returnează *true*/1 sau *false*/0 în funcție de situație (dacă $x[k] \neq x[k-1]$, respectiv $x[k] = x[k-1]$). Pentru a nu avea un caz particular $k=1$, înainte de apelul `generare(1)` se face inițializarea lui $x[0]$ cu un caracter care nu este în alfabet (de exemplu $x[0]='*'$).

Trecerea la metoda backtracking

Folosind considerațiile anterioare putem să discutăm despre rezolvarea unei probleme generale.

Metoda backtracking permite să se rezolve probleme de tipul:

Se dau n mulțimi A_1, A_2, \dots, A_n și o proprietate P , care depinde de x_1, x_2, \dots, x_n (x_1 din A_1 , x_2 din A_2 , ..., x_n din A_n). Se cere să se determine toate șirurile x_1, x_2, \dots, x_n , care verifică proprietatea P .

Particularizare

Pentru problema cu generarea cuvintelor în care literele de pe poziții consecutive sunt distincte, avem:

A_1, A_2, \dots, A_n sunt toate formate din primele m litere mici ale alfabetului englez.

P este condiția $x_i \neq x_{i-1}$, i din mulțimea $\{2, 3, \dots, n\}$.

Descrierea metodei de rezolvare (numită backtracking datorită mecanismului de generare a soluțiilor)

- Construirea componentelor vectorului x se va face în ordinea crescătoare a indicilor, $x_1, x_2, \dots, x_k, \dots, x_n$. x_k fiind din mulțimea A_k .

- Din condiția P se deduc condiții parțiale, pentru componentele x_1, x_2, \dots, x_k . Acestea vor fi notate cu $P(k)$.

- Ideea metodei constă în faptul că dacă, la un moment dat trebuie să se construiască x_k , se consideră pe rând elementele lui A_k și dacă sunt verificate condițiile $P(k)$, atunci se trece la următoarea componentă, altfel se merge înapoi la componenta anterioară și se caută altă valoare. În permanență există un “dute-vino” (realizat de mecanismul recursivității) până se ajunge la câte o soluție.

- Forma generală a unui subprogram recursiv de generare a vectorului x este prezentată în continuare.

<pre> procedure back(k:integer); var i:Tip; begin if k=n+1 then afisare else for i din A_k do begin x[k]:=i; if P(k) then back(k+1) end end; </pre>	<pre> void back(int k){ Tip i; if (k==n+1) afisare(); else for (i din A_k) { x[k]=i; if (P(k)) back(k+1); } } </pre>
--	---

- Subprogramul de generare trebuie apelat prin **back(1)**, pentru a construi vectorul x începând cu prima componentă.

Observații

- Metoda backtracking determină toate soluțiile problemei.
- Cu cât condițiile parțiale $P(k)$ sunt mai restrictive cu atât timpul de execuție este mai mic.
- Uneori este de preferat să folosim în componentele lui x , indicii elementelor din mulțimile A_1, A_2, \dots, A_n , pentru că aceștia sunt numere consecutive și pot fi parcurși mai ușor.
- Timpul de execuție al algoritmilor ce folosesc metoda backtracking este exponențial (relativ la n).
- Dacă există alte metode de rezolvare cu timp de execuție polinomial, atunci această metodă poate fi folosită doar pentru a confirma corectitudinea celeilalte, prin compararea rezultatelor.
- Pentru a însuși această metodă de programare este nevoie de rezolvarea unui număr mare de probleme, începând cu cele clasice (problema damelor, colorarea unei hărți, generarea elementelor combinatoriale, plata unei sume de bani, etc.), apoi continuând cu cele care necesită modificări ale subprogramului de generare.
- Metoda backtracking poate fi utilizată și pentru generări de lanțuri în tablouri bidimensionale (așa numitul *backtracking în plan*) însă pentru această variantă ar trebui să scris un alt material.

Probleme propuse

1. Se dă o mulțime A cu n ($n < 20$) elemente numere naturale < 32000 . Se cere să se determine toate modalitățile de împărțire a lui A în trei mulțimi cu aceeași sumă a elementelor.

Exemplu

multime.in	multime.out
9	{7 3 6 4} {2 8 10} {11 9}
2 8 7 11 6 4 9 3 10	...

2. Se dă un șir cu n ($n < 50$) elemente numere naturale < 100 . Se cere să se determine toate numere din șir, care se pot scrie ca sumă de numere prime distincte.

Exemplu

sir.in	sir.out
3 6 7 10	7 10

3. Se dă n număr natural ($n < 20$). Determinați toate numerele în baza 2 cu n cifre, care au numărul de cifre de 1 egal cu cel al celor de 0.

Exemplu

baza2.in	baza2.out
4	1100 1010 1001

4. Se dă un șir cu n ($n < 20$) elemente, numere naturale < 200 distincte. Se cere să se determine cel mai mic număr natural, care nu poate fi scris ca o sumă de termeni din șirul dat.

Exemplu

mic.in	mic.out
4 1 2 4 10	8

5. Se dă un șir cu n ($n < 20$) elemente, numere naturale < 200 distincte. Se cere să se determine toate subșirurile crescătoare de lungime maximă ale șirului dat.

Exemplu

subsir.in	subsir.out
7 8 3 5 9 2 3 4	3 5 9 2 3 4

6. Se dă un șir cu n ($n < 20$) elemente, numere întregi cu valoarea absolută < 200 . Se cere să se determine toate valorile epresiilor aritmetice ce se pot obține punând între orice două numere vecine operatorul $+$ sau $-$.

Exemplu

exp.in	exp.out	Explicatie
3 2 8 -2	8 12 -8 -4	2+8+(-2) 2+8-(-2) 2-8+(-2) 2-8-(-2)